**Полиморфизъм (Polymorphism)**

Следващият основен принцип от обектно-ориентираното програмиране е "полиморфизъм".**Полиморфизмът** позволява третирането на обекти от наследен клас като обекти от негов базов клас. Например големите котки (базов клас) хващат жертвите си (метод) по различен начин. Лъвът (клас наследник) ги дебне, докато Гепардът (друг клас-наследник) просто ги надбягва.

Полиморфизмът дава възможността да третираме произволна голяма котка просто като голяма котка и да кажем "хвани жертвата си", без значение каква точно е голямата котка.

Полиморфизмът може много да напомня на абстракцията, но в програми­рането се свързва най-вече с пренаписването (override) на методив нас­ледените класове с цел промяна на оригиналното им поведение, насле­дено от базовия клас. Абстракцията се свързва със създаването на интерфейс на компонент или функционалност (дефиниране на роля). Пренаписва­нето на методи ще разгледаме в детайли след малко.

### Абстрактни класове

Какво става, ако искаме да кажем, че класът **Felidae** е непълен и само наследниците му могат да имат инстанции? Това става с ключовата дума **abstract** пред името на класа и означава, че класът не е готов и не може да бъде инстанциран. Такъв клас се нарича **абстрактен клас**. А как да укажем коя точно част от класа не е пълна? Това отново става с ключовата дума **abstract** пред името на метода, който трябва да бъде имплементиран. Този метод се нарича **абстрактен метод** и не може да притежава имплементация, а само декларация.

Всеки клас, който има поне един абстрактен метод, трябва да бъде абстрактен. Логично, нали? Обратното, обаче не е в сила. Възможно е да дефинираме клас като абстрактен дори когато в него няма нито един абстрактен метод.

Абстрактните класове са нещо средно между клас и интерфейс. Те могат да дефинират обикновени методи и абстрактни методи. Обикновените методи имат тяло (имплементация), докато абстрактните методи са празни (без имплементация) и са оставени да бъдат реализирани от класовете-наследници.

#### Абстрактен клас – примери

Да разгледаме един пример за абстрактен клас:

|  |
| --- |
| **Felidae.cs** |
| /// <summary>/// Latin word for "cat"/// </summary>public abstract class Felidae{      // ...       protected void Hide()      {            // ...      }       protected void Run()      {            // ...      }       public abstract bool CatchPray(object pray);} |

Забележете в горния пример как нормалните методи **Hide()**и **Run()**имат тяло, а абстрактният метод **CatchPray()** няма тяло. Забележете, че методите са **protected**.

Ето как изглежда имплементацията:

|  |
| --- |
| **Lion.cs** |
| public class Lion : Felidae, Reproducible<Lion>{      protected void Ambush()      {            // ...      }       public override bool CatchPray(object pray)      {            base.Hide();            this.Ambush();            base.Run();            // ...            return false;      }} |

Ето още един пример за абстрактно поведение, реализирано чрез абстрак­тен клас и полиморфно извикване на абстрактен метод. Първо дефини­раме абстрактния клас **Animal**:

|  |
| --- |
| **Animal.cs** |
| public abstract class Animal{      public void PrintInformation()      {            Console.WriteLine("I am {0}.", this.GetType().Name);            Console.WriteLine(GetTypicalSound());      }       protected abstract String GetTypicalSound();} |

Дефинираме и класа **Cat**, който наследява абстрактния клас **Animal** и дефинира имплементация за абстрактния метод **GetTypicalSound()**:

|  |
| --- |
| **Cat.cs** |
| public class Cat : Animal{      protected override String GetTypicalSound()      {            return "Miaoooow!";      }} |

Ако изпълним следната програма:

|  |
| --- |
| public class AbstractClassExample{      public static void Main()      {            Animal cat = new Cat();            cat.PrintInformation();      }} |

... ще получим следния резултат:

|  |
| --- |
| **I am Cat.****Miaoooow!****Press any key to continue . . .** |

В примера методът **PrintInformation()** от абстрактния клас свършва своята работа като разчита на резултата от извикването на абстрактния метод **GetTypicalSound()**, който се очаква да бъде имплементиран по различен начин за различните животни (различните наследници на класа **Animal**). Различните животни издават различни звуци, но отпечатването на информация за животно е една и съща функционалност за всички животни и затова е изнесена в базовия клас.

#### Чист абстрактен клас

Абстрактните класове, както и интерфейсите не могат да се инстанцират. Ако се опитате да създадете инстанция на абстрактен клас, ще получите грешка по време на компилация.

|  |  |
| --- | --- |
| clip_image001[7] | **Понякога даден клас може да бъде деклариран като абстрактен дори и да няма нито един абстрактен метод, просто, за да се забрани директното му използване, без да се създава инстанция на негов наследник.** |

**Чист абстрактен клас (pure abstract class)** е абстрактен клас, който няма нито един имплементиран метод, както и нито една член промен­лива. Много напомня на интерфейс. Основната разлика е, че един клас може да имплементира много интерфейси и наследява само един клас (бил той и чист абстрактен клас).

В началото при съществуването на множествено наследяване не е имало нужда от интерфейси. За да бъде заместено, се е наложило да се появят интерфейсите, които да носят многото роли на един обект.

### Виртуални методи

Метод, който може да се пренапише в клас наследник, се нарича **виртуален метод (virtual method)**. Методите в .NET не са виртуални по подразбиране. Ако искаме да бъдат виртуални, ги маркираме с ключовата дума **virtual**. Тогава клас-наследник може да декларира и дефинира метод със същата сигнатура.

Виртуалните методи са важни за **пренаписването на методи** **(method overriding)**, което е в сърцето на полиморфизма.

#### Виртуални методи – пример

Имаме клас, наследяващ друг, като и двата имат общ метод. И двата метода пишат на конзолата. Ето как изглежда класът **Lion**:

|  |
| --- |
| **Lion.cs** |
| public class Lion : Felidae, Reproducible<Lion>{      public override void CatchPray(object pray)      {            Console.WriteLine("Lion.CatchPray");      }} |

Ето как изглежда и класът **AfricanLion**:

|  |
| --- |
| **AfricanLion.cs** |
| public class AfricanLion : Lion{      public override void CatchPray(object pray)      {            Console.WriteLine("AfricanLion.CatchPray");      }} |

Правим три опита за създаване на инстанции и извикване на метода **CatchPray**.

|  |
| --- |
| **VirtualMethodsExample.cs** |
| public class **VirtualMethodsExample**{      public static void Main()      {            {                  Lion lion = new Lion(true, 80);                  lion.CatchPray(null);                  // Will print "Lion.CatchPray"            }             {                  AfricanLion lion = new AfricanLion(true, 120);                  lion.CatchPray(null);                  // Will print "AfricanLion.CatchPray"            }             {                  Lion lion = new AfricanLion(false, 60);                  lion.CatchPray(null);                  // Will print "AfricanLion.CatchPray", because                  // the variable lion has value of type AfricanLion            }      }} |

В последния опит ясно се вижда как всъщност се извиква пренаписаният метод, а не базовият. Това се случва, защото се проверява кой всъщност е истинският клас, стоящ зад променливата, и се проверява дали той има имплементиран (пренаписан) този метод.

Пренаписването на методи се нарича още: **препокриване (подмяна) на виртуален метод**.

Както виртуалните, така и абстрактните методи могат да бъдат препокри­вани. Абстрактните методи всъщност представляват виртуални методи без конкретна имплементация. Всички методи, които са дефинирани в даден интерфейс са абстрактни и следователно виртуални, макар и това да не е дефинирано изрично.

#### Виртуални методи и скриване на методи

В горния пример имплементацията на базовия клас остана скрита и неиз­ползвана. Ето как можем да ползваме и нея като част от новата имплемен­тация (в случай че не искаме да подменим, а само да допълним старата имплементация).

Ето как изглежда и класът **AfricanLion**:

|  |
| --- |
| **AfricanLion.cs** |
| public class AfricanLion : Lion{      public override void CatchPray(object pray)      {            Console.WriteLine("AfricanLion.CatchPray");            Console.WriteLine("calling base.CatchPray");            Console.Write("\t");            base.CatchPray(pray);            Console.WriteLine("...end of call.");      }} |

В този пример при извикването на **AfricanLion.catchPray(…)** ще се изпишат 3 реда на конзолата:

|  |
| --- |
| **AfricanLion.CatchPray****calling base.CatchPray****Lion.CatchPray****...end of call.** |

#### Разликата между виртуални и невиртуални методи

Някой може да попита каква е разликата между виртуалните и невиртуал­ните методи.

Виртуални методи се използват, когато очакваме наследяващите класове да променят/допълват/изменят дадена функционалност. Например мето­дът **Object.ToString()** позволява наследяващите класове да променят както си искат имплементацията. И тогава дори когато работим с един обект не директно, а чрез upcast до object пак използваме пренаписаната имплементация на виртуалните методи.

Виртуалните методи са ключова способност на обектите когато говорим за [абстракция](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane/#_Абстракция_(Abstraction)) и работа с абстрактни типове.

Запечатването на методи (**sealed**) се прави, когато разчитаме на дадена функционалност и не желаем тя да бъде променяна. Разбрахме, че методите по принцип са запечатани. Но ако искаме един виртуален метод от базов клас да запечатаме в класа наследник, използваме **override sealed**.

Класът **string** няма нито един виртуален метод. Всъщност наследяването на **string** е забранено изцяло с ключовата дума **sealed** в декларацията на класа. Ето част от декларацията на **string** и**object** (триеточието в квадратните скоби указва пропуснат код, който не е релевантен):

|  |
| --- |
| namespace System{      [...] public class Object      {            [...] public Object();             [...] public virtual bool Equals(object obj);            [...] public static bool Equals(object objA, object objB);            [...] public virtual int GetHashCode();            [...] public Type GetType();            [...] protected object MemberwiseClone();            [...] public virtual string ToString();      }        [...] public sealed class String : [...]      {            [...] public String(char\* value);             [...] public int IndexOf(string value);            [...] public string Normalize();            [...] public string[] Split(params char[] separator);            [...] public string Substring(int startIndex);            [...] public string ToLower(CultureInfo culture);             [...]      }} |

### Интерфейси

В езика C# **интерфейсът** е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение.

Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки.

Например един обект **Човек** може да има ролите **Военен** (с поведение "стреляй по противника"),**Съпруг** (с поведение "обичай жена си"), **Данъкоплатец** (с поведение "плати си данъка"). Всеки човек обаче импле­ментира това поведение по различен начин: **Иван** си плаща данъците навреме, **Георги** – не навреме, **Петър** – въобще не ги плаща.

Някой може да попита защо най-базовият за всички обекти клас **Object** не е всъщност интерфейс. Причината е, че тогава всеки клас щеше да трябва да имплементира една малка, но много важна, група методи, а това би отнемало излишно време. Оказва се, че и не всеки клас има нужда от специфична реализация на **Object.GetHashCode()**, **Object.Equals(…)**, **Object.ToString()**, тоест имплемен­тацията по подразбиране върши работа в повечето случаи. От класа **Object** не е нужно да се пренапише (повторно имплементира) никой метод, но ако се наложи, това може да се направи. Пренаписването на методи е обяснено в детайли в секцията за [виртуални методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane/#_Виртуални_методи).

#### Интерфейси – ключови понятия

В интерфейса може да има само декларации на методи и константи.

**Сигнатура на метод (method signature)** е съвкупността от името на метода + описание на параметрите (тип и последователност). В един клас/интерфейс всички методи трябва да са с различни сигнатури и да не съвпадат със сигнатури на наследени методи.

**Декларация на метод (method declaration)** е съвкупността от връщания тип на метода + сигнатурата на метода. Връщаният тип е просто за яснота какво ще върне метода.

|  |  |
| --- | --- |
| clip_image001[6] | **Това, което идентифицира един метод, е неговата сигна­тура. Връщаният тип не е част нея. Причината е, че ако два метода се различават само по връщания тип (напри­мер два класа, които се наследяват един друг), то не може еднозначно да се идентифицира кой метод трябва да се извика.** |

**Имплементация на клас/метод (class/method implementation)** е тялото със сорс код на класа/метода. Най често е заключено между скобите **{** и **}**. При методите се нарича още **тяло на метод**.

#### Интерфейси – пример

Интерфейсът в .NET се дефинира с ключовата думичка **interface**. В него може да има само декларации на методи, както и статични променливи (за константи например). Ето един пример за интерфейс:

|  |
| --- |
| **Reproducible.cs** |
| public interface Reproducible<T> where T:Felidae{      T[] Reproduce(T mate);} |

За шаблонни типове (Generics) сме говорили в главата "[Дефиниране на класове](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove)". Интерфейсът, който сме написали, има един метод от тип **Т** (**Т** трябва да наследява **Felidae**) и връща масив от **Т**.

Ето как изглежда и класът **Lion**, който имплементира интерфейса **Reproducible**:

|  |
| --- |
| **Lion.cs** |
| public class Lion : Felidae, Reproducible<Lion>{      // ...       Lion[] Reproducible<Lion>.Reproduce(Lion mate)      {            return new Lion[]{new Lion(true, 12), new Lion(false, 10)};      }} |

Името на интерфейса се записва в декларацията на класа (първия ред) и се специфицира шаблонният клас.

Можем да укажем метод на кой интерфейс имплементираме, като му напишем името:

|  |
| --- |
| Lion[] Reproducible<Lion>.Reproduce(Lion mate) |

В интерфейса методите само се декларират, имплементацията е в класа, който имплементира интерфейса – **Lion**.

Класът, който имплементира даден интерфейс, трябва да имплементира всеки метод от него. Изключение – ако класът е абстрактен, тогава може да имплементира нула, няколко или всички методи. Всички останали методи се имплементират в някой от класовете наследници.

#### Абстракция и интерфейси

Най-добрият начин да се реализира абстракция е да се работи с интер­фейси. Един компонент работи с интерфейси, които друг имплементира. Така подмяната на втория компонент няма да се отрази на първия, стига новият компонент да имплементира старите интерфейси. Интерфейсът се нарича още**договор** (**contract**). Всеки компонент, имплементирайки един интерфейс, спазва определен договор (сигнатурата на методите). Така два компонента, стига да спазват правилата на договора, могат да общуват един с друг, без да знаят как работи другата страна.

Примери за важни интерфейси от Common Type System (CTS) са**System.Collections.Generic.IList<T>** и **System.­Collections.­Generic. ICollection<T>**. Всички стандартни колекции импле­мен­тират тези интерфейси и различните компоненти си прехвърлят различни имплементации (масиви или свързани списъци, хеш-таблици, червено-черни дървета и др.) винаги под общ интерфейс.

Колекциите са един отличен пример на обектно-ориентирана библиотека с класове и интерфейси, при която се използват много активно всички основни принципи на ООП: абстракция, наследяване, капсулация и поли­морфизъм.

#### Кога да използваме абстракция и интерфейси?

Отговорът на този въпрос е: винаги, когато искаме да постигнем абстрак­ция на данни или действия, чиято имплементация по-късно може да се подмени. Код, който комуникира с друг код чрез интерфейси е много по-издръжлив срещу промени, отколкото код, напи­сан срещу конкретни кла­сове. Работата през интерфейси е често сре­щана и силно препоръчвана практика – едно от основните правила за писане на качествен код.

#### Кога да пишем интерфейси?

Винаги е добра идея да се използват интерфейси, когато се предоставя функционалност на друг компонент. В интерфейса се слага само функцио­налността (като декларация), която другите трябва да виждат.

Вътрешно в една програма/компонент интерфейсите могат да се използват за дефиниране на роли. Така един обект може да се използва от много класове чрез различните му роли.